
NIST Smart Flow System

User's guide

Christelle Martin
Olivier Galibert, Martial Michel, Fabrice Mougin, Vincent Stanford

<http://www.nist.gov/smartspace/>

National Institute of Standards and Technology, USA

Version 3.7, Last updated: December 13, 2000

Contact us:

Vincent Stanford - Vincent.Stanford@nist.gov
N.I.S.T.
Bldg 220, Rm A231
100 Bureau Drive, Stop 8940
Gaithersburg, MD 20899-8940 - USA

Contents

1	Introduction	1
1.1	The NIST Smart Space project	1
1.2	About this document	2
2	Concept	3
2.1	Introduction	3
2.1.1	Illustration	3
2.1.2	Main terms	3
2.2	Flows	5
2.2.1	Identification	5
2.2.2	Synchronous and asynchronous flows	5
2.2.3	Optional flow	6
2.3	How to use a data flow	7
2.3.1	Opening a flow in emit mode	7
2.3.2	Subscribing to a flow in receive mode	7
2.3.3	Example	7
3	Profile / Description of sflib.h file	9
3.1	Types	9
3.1.1	Basic types	9
3.1.2	Parameter types	10
3.1.3	Metadata types	10
3.2	Basic functions	11
3.2.1	<code>sf_init_param_xxx</code> - accessors	11
3.2.2	<code>sf_init_param_set_flow_xxx</code>	11
3.2.3	<code>sf_init_param_get_flow_xxx</code>	12
3.2.4	<code>sf_flow_param_init</code>	13
3.2.5	<code>sf_flow_param_get_flow</code>	13
3.2.6	<code>sf_flow_param_set_xxx</code> - general	13
3.2.7	<code>sf_flow_param_set_xxx</code> - Video flow	14
3.2.8	<code>sf_flow_param_set_xxx</code> - Audio flow	14
3.2.9	<code>sf_flow_param_set_xxx</code> - Data flow	15
3.2.10	<code>sf_flow_param_set_xxxx</code> - Vector flow	15
3.2.11	<code>sf_flow_param_set_xxx</code> - Matrix flow	15
3.2.12	<code>sf_init</code>	16
3.2.13	<code>sf_flow_subscribe_xxx</code>	16
3.2.14	<code>sf_flow_create</code>	16
3.2.15	<code>sf_get_output_bufferxxx</code>	17
3.2.16	<code>sf_send_buffer</code>	17
3.2.17	<code>sf_drop_buffer</code>	17

3.2.18	<code>sf_get_update_buffer</code>	18
3.2.19	<code>sf_wait_update</code>	18
3.2.20	<code>sf_get_buffer</code>	18
3.2.21	<code>sf_get_buffer_size</code>	19
3.2.22	<code>sf_release_buffer</code>	19
3.2.23	<code>sf_perror</code>	20
3.2.24	<code>sf_flow_xxx_metadata</code>	20
3.2.25	<code>sf_set_blocking_mode</code>	21
3.2.26	<code>sf_flow_xxx_close & sf_exit</code>	21
3.3	Example of use	22
3.3.1	Type definition	22
3.3.2	Initialize the system parameters	22
3.3.3	Initialize the system	22
3.3.4	Initialize the flow parameters	23
3.3.5	Open a flow in emit mode and emit on it	23
3.3.6	Subscribe to the flow and read its data	24
3.3.7	Close the flow and quit the system	25
3.4	Example	26
3.4.1	Video capture: <code>vcap.c</code>	26
3.4.2	Video display: <code>vdisp.c</code>	27
4	Conclusion	31

List of Figures

2.1	Communication system	4
2.2	Flow	5
2.3	Camera group	7

Chapter 1

Introduction

1.1 The NIST Smart Space project

The NIST Smart Space Laboratory has been created to offer assistance to industrial research and product development laboratories as to face the numerous performance and interoperability challenges inherent in the Smart Work Spaces of the future. We believe that the shift to Pervasive Computing is already well under way and will have as much impact on industry, and daily life as personal computing did. Our Modular Test Bed is designed to allow our industrial and academic laboratories to bring their technologies together for integration and performance testing needed.

Pervasive Computing refers to the trend to numerous, easily accessible computing devices connected to each other and to an ubiquitous network infrastructure. This will create new opportunities and challenges for IT companies as they place computers and sensors in devices, appliances, and equipment in buildings, homes, workplaces, and factories. Within a few years, embedded devices able to execute complex software applications, and use wireless communications will be the norm and their effective use requires distributed interfaces on numerous, small, and even invisible devices.

We are prototyping an experimental smart space data-flow system. The prototype focus is on advanced forms of human-computer-interaction. Integrating wireless networks with dynamic service discovery, automatic device configuration, and sensor based perceptual interfaces. The objectives are to facilitate:

- Identify security mechanisms needed to ensure privacy, integrity, and accessibility of implementations
- Develop metrics, test methods, and standard reference data sets to pull the technology forward
- Provide reference implementations to serve as models for possible commercial implementations
- Interconnect the prototype components and systems to explore key issues associated with distributed smart spaces
- Establish an integrated multi-sensor perceptual interface test bed for smart work spaces
- Support integration of the AirJava/Aroma components
- Deploy perceptual interface components from our industrial advisors to investigate system level performance and metrics issues
- Provide a multi-sensor data recording environment for the production of standard test materials

We have developed the Smart Space Modular Test bed which consists of a defined middle-ware API for real-time data transport, and a connection broker server for sensor data sources and processing data sinks. For example, a microphone array acquires a speech signal, reduces it to a single channel, and offers it as a data-flow. Then a speaker identification system subscribes to the signal flow, while a speaker-dependent speech recognition system subscribes as well. The speaker ID system then offers a real-time flow, to which the speech recognition system also subscribes. This layer makes it possible if not necessary easy to integrate components that were not intentionally designed to work together, such as speaker identification, and speech recognition systems. Many constituent technologies are under separate development in industry, so the issues of interoperability and integration are paramount. This project will develop metrics and reference material based on real implementations for industrial use. Important technologies include is sensor-based collaborative interfaces using:

- Speech recognition
- Speaker identification
- Face recognition
- Source localization/separation
- Channel normalization
- Immersive video and acoustic displays
- Personal information appliances
- Pervasive networking
- Information storage and retrieval
- multimedia data types
- Multiple Interconnected Spaces

To foster integration, interoperability, and the development of emerging technologies, standardization and measurements are critical but for economic reasons, often are not addressed by individual companies. The lack of common software infrastructure, tools to create, manage, measure, test, and debug pervasive services, standards in key areas such as service discovery, APIs, wireless networks, automatic configuration, and ad hoc transactional security, all currently impede widespread adoption of pervasive computing. Also, measurements, and tests developed and performed in a public forum allow competing research systems to be compared, and improved systems to build on the best features of previous iterations.

It is a long-term research platform that will provide a sensor-rich collaborative working environment. Sensor technologies include microphone arrays, video camera arrays, smart badges, I.R. room scanners, and possibly person position sensors. The major component areas will include:

1.2 About this document

In this document, we introduce you to NIST Smart Flow System and help you getting started. It is dedicated to users, people who have to bring and integrate their part into the platform, without modifying the code of this platform. This user's guide is of course also useful for programmers, those whose job is actually to intervene in the system code.

The communication architecture has been implemented for the Linux Operating System, using the C/C++ programming languages. For this reason the User's guide is accessible to anyone having a basic programming background. However, the Programmer's guide requires knowledge in data processing communication system.

Chapter 2

Concept

2.1 Introduction

In this section, we introduce you to main concepts, terms and components architecture.

2.1.1 Illustration

Figure 2.1 shows a distributed system using data-flows to communicate between data capture components and data analysis components. These components then use the flow system to offer coordinates of the face and the spoken text to other processes in the Smart Space.

2.1.2 Main terms

We will now define the important terms and concepts needed for future discussion.

Data-flow

The NIST Smart Flow System uses data-flows to convey information. As shown in figure 2.2, a data-flow transports a stream of data from one client (the emitter) to one or more other clients (the subscribers),

Data-flow can be for example:

- pictures, or digital video camera images stream
- microphone audio signals

In figure 2.1 the Video in flow is distributed from Face Localization client and to Face Recognition client.

Client

A client is a program that uses the flows to obtain, produce and/or process data.

Examples of clients (see figure 2.1):

- Face Recognition client uses Video in flow
- Audio Capture client uses Audio flow

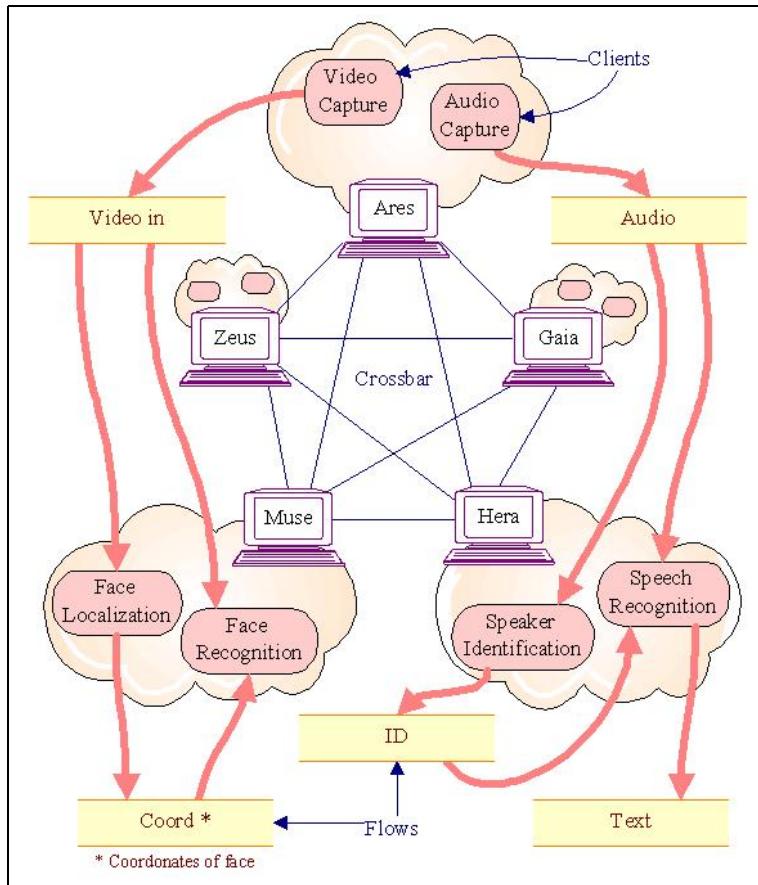


Figure 2.1: Overview of the system: Clients are linked to servers and communicate each other using data-flows.

Server

A server provides flows with data transport communication, dispatching their data.

Examples of servers (see figure 2.1):

- Muse's server enables Face Recognition to communicate,
- Cyclops' server enables Audio Capture to communicate.

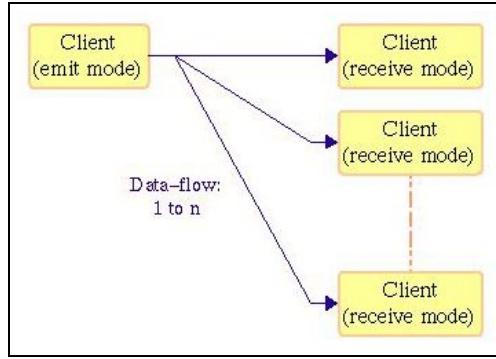


Figure 2.2: Flows dispatch information from one client to n

2.2 Flows

2.2.1 Identification

Components are gathered together according to their physical device. For instance (see figure 2.3), Video Capture and Face Localization belong to the group called Camera. However, the system can have several identical physical devices (several cameras for instance). We could then find several flows having a same name whereas they are not conveying the same information. We distinguish them via the physical device with a structure called group.

Thus, a system with two cameras has two groups relative to each of them: Camera 1 and Camera 2. Both groups use a Video in flow. One Video in flow belongs to Camera 1 group, the other Video in flow to Camera 2 group.

These two information attributes: flow name and flow group, are needed to uniquely identify a flow. Nevertheless, another required information attribute is the type of the conveyed data. Type of data can be *Video* ("mjpeg", "packed rgb", "720*480*24), *Audio*, *Data*, *Vector* or *Matrix*. *Vector* flows convey fixed-size-vectors the type of which can be automatically converted by a server (float, double, 16bits).

Finally, a data-flow is defined by its name, group and data type. The default group will be the group the component belongs to.

This is an example of a defined data-flow:

- Name: *Video in*
- Group: *Camera 1*
- Type: *Video*

The key to uniquely identify this flow would be *Video in* flow and *Camera1* group.

2.2.2 Synchronous and asynchronous flows

Data-flows are structured in buffers.

For instance,

- an image for a *Video* flow,
- a hundredth of second for an *Audio* flow.

Synchronous flows

These data-flows are synchronized by themselves.

The client asks for a particular buffer (for example: the last arrived) and blocks until it gets it. So, a synchronous flow stays stable while the client processes its data. Hence, it is possible to go back in the past and to be sure not to miss any buffers.

Asynchronous flows

Asynchronous data-flows are not implemented. This is request of comment, please feel free to send us your comments about asynchronous flows.

The buffer content of an asynchronous flow is automatically updated, without any user intervention.

An asynchronous flow is used when the most recent state is required, but not the previous states or the time of the last update. Nevertheless, a client can get the time of the last update, and also block until the next update.

This flow would for example be used to give the current position of a camera.

2.2.3 Optional flow

A parameter called `active` makes it possible to use optional flows. A flow that is not used by default in an application, can still be created when the application is launched if its name is in the command line parameters.

A flow is by default active (`active = 1`), unless it is asked to be inactive (`active = 0`), it is then an optional flow.

The user can activate an inactive flow, but it is impossible to deactivate an active flow.

Page [27](#) shows an example with `vdisp.c` - code that uses this concept.

2.3 How to use a data flow

A client can be either an emitter or a receptor. It can use a flow in three different ways to:

- open a flow in emit mode (it emits information on this flow),
- subscribe to a synchronous flow in receive mode (it receives information from a synchronous flow), or
- subscribe to an asynchronous flow in receive mode (it receives information from an asynchronous flow).

2.3.1 Opening a flow in emit mode

To open a flow in emit mode, a client first creates a flow and then writes data on it.

```
Create a flow
Loop (
    Allocate a buffer
    Write data in the buffer
    Send the buffer
)
```

2.3.2 Subscribing to a flow in receive mode

To open a flow (synchronous or asynchronous) in receive mode, a client subscribes to it, reads each buffer and processes its data before releasing it.

```
Subscribe to the flow
Loop (
    Read the buffer
    Process its data
    Release the buffer
)
```

2.3.3 Example

Consider for instance the Camera group, displayed in figure 2.3.

Video capture, Face localization and Face recognition are three clients that belong to this group.

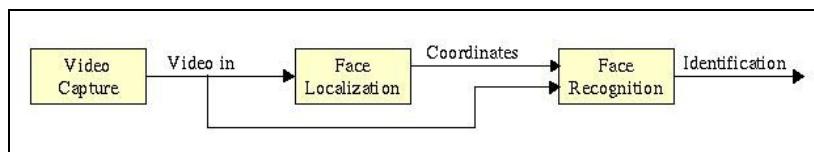


Figure 2.3: Camera group: its clients and the flows they use

These Camera group clients provide information to data-flows, and/or receive information from flows:

- Capture emits on Video in flow,

- Face Localization emits on Coordinates flow,
- Face Recognition emits on Identification flow.
- Face Localization subscribes to Video in flow,
- Face Recognition subscribes to both Video in and Coordinates flows.

Chapter 3

Profile / Description of sflib.h file

This chapter introduces you to `sflib.h`. You will find a presentation of its main types and functions. Then, it will describe their use, and illustrate these concepts by an example.

The `sflib.h` file contains the definitions of important types, subtypes, constants and parameters.
N.B.: `xxx` stands for any string, and `sf` means smart flow.

3.1 Types

`sflib.h` defines three kind of types:

- basic types,
- parameter types,
- metadata types.

3.1.1 Basic types

`sf_handle`

`sf_handle` is a structure that contains the information used by the library to communicate with the server. The initialization call of the library allocates one of these structures and gives back a pointer to it. This pointer becomes a parameter for all following library calls.

`sf_flow_sync`

`sf_flow_sync` is a structure that contains the information used by the library to communicate with the server, and stands for a flow the client has subscribed to with synchronous updating.

`sf_flow_sync` is used instead of `sf_handle` for all calls relative to this flow.

`sf_flow_async`

`sf_flow_async` is similar to `sf_flow_sync`, but for an asynchronous data flow.

`sf_flow_emit`

`sf_flow_emit` is similar to `sf_flow_sync`, but concerning a flow opened in emit mode.

sf_counter

`sf_counter` is an integer type corresponding to the buffer number.

sf_error

`sf_error` is a generic structure used to return the error conditions when an error occurs.

3.1.2 Parameter types

Parameter types are opaque structures that describe an application or a flow.

sf_init_param

`sf_init_param` holds an application parameters. It is used by all `sf_init_param_xxx` functions.

sf_flow_param

`sf_flow_param` holds the flow parameters. It used by all `sf_flow_param_xxx` functions

3.1.3 Metadata types

sf_typeinfo_data

`sf_typeinfo_data` describes data presentation. It holds their size and name.

sf_typeinfo_flow

`sf_typeinfo_flow` describes the global type of an *Audio*, *Video*, *Data*, *Vector*, *Matrix* data-flow or and holds information about it:

- for a *Video* flow: image size (width and height)
- for an *Audio* flow: sound frequency
- for a *Data* flow: data size and name
- for a *Vector* flow: vector size
- for a *Matrix* flow: matrix size

3.2 Basic functions

3.2.1 sf_init_param_xxx - accessors

- `sf_init_param_init` initializes the application structure:
`void sf_init_param_init(sf_init_param *param);`
- `sf_init_param_set_name` sets the application logical name:
`void sf_init_param_set_name(sf_init_param *param, const char *name);`
- `sf_init_param_set_group` sets the application group (optional):
`void sf_init_param_set_group(sf_init_param *param, const char *group);`
- `sf_init_param_args` sets the configuration structure according to `argc` and `argv`. This is the last function of this set to be called since it uses the parameters previously set. `argc` and `argv` are updated to remove the parsed parameters.
`void sf_init_param_args(sf_init_param *param, int *argc, char **argv);`

These functions set the following parameters:

- `sf-appname`: application name
- `sf-group`: group
- `sf-flow`: (i or o, flow number, name, group) for an output flow (o) or an input flow (i); group is optional
- `sf-active`: (i or o, flow number) activates a flow
- `sf-dump-args`: this one is internal and gathers a flow description

Returned value

none

Parameters

- `param`: configuration structure
- `name`: application logical name (e.g. Face recognition)
- `group`: application group (e.g. Camera 1)
- `argc` and `argv`: command line parameters

3.2.2 sf_init_param_set_flow_xxx

- `sf_init_param_set_flow_count` sets the number of input and output flows:
`void sf_init_param_set_flow_count(sf_init_param *param, int input, int output);`
- `sf_init_param_set_flow_name` sets the flow name:
`void sf_init_param_set_flow_name(sf_init_param *param, sf_flow_kind kind, int number, const char *name);`
- `sf_init_param_set_flow_group` sets the flow group:
`void sf_init_param_set_flow_group(sf_init_param *param, sf_flow_kind kind, int number, const char *group);`

- `sf_init_param_set_flow_type` sets the flow type:
`void sf_init_param_set_flow_type(sf_init_param *param, sf_flow_kind kind, int number, sf_flow_type type);`
- `sf_init_param_set_flow_active` sets the flow active or inactive:
`void sf_init_param_set_flow_active(sf_init_param *param, sf_flow_kind kind, int number, int active);`
- `sf_init_param_set_flow_desc` sets the of a data flow description:
`void sf_init_param_set_flow_desc(sf_init_param *param, sf_flow_kind kind, int number, DATA_DESC_P);`
- `sf_init_param_set_flow_width` sets the vector size of a vector flow:
`void sf_init_param_set_flow_width(sf_init_param *param, sf_flow_kind kind, int number, int width);`

Returned value

none

Parameters

- `param`: configuration structure
- `input`: number of input flows
- `output`: number of output flows
- `kind`: SF_KIND_INPUT for an input flow or SF_KIND_OUTPUT for an output flow
- `number`: flow number
- `width`: vector width
- `name`: flow name
- `group`: flow group
- `type`: SF_TYPE_AUDIO, SF_TYPE_VIDEO, SF_TYPE_DATA, SF_TYPE_VECTOR, SF_TYPE_MATRIX
- `DATA_DESC_P`: DATA_DESC(type) or DATA_DESC_WITH_SIZE(type, size)
- `active`: is 1 when the flow is active (default value), and 0 when it is not (optional flow)

3.2.3 `sf_init_param_get_flow_xxx`

- `sf_init_param_get_flow_active` tests if a flow is active:
`int sf_init_param_get_flow_active(sf_init_param *param, sf_flow_kind kind, int number);`
- `sf_init_param_get_flow_name` gets the flow name:
`const char *sf_init_param_get_flow_name(sf_init_param *param, sf_flow_kind kind, int number);`
- `sf_init_param_get_flow_group` gets the flow group:
`const char *sf_init_param_get_flow_group(sf_init_param *param, sf_flow_kind kind, int number);`

Returned value

- `sf_init_param_get_flow_active`: 0 if there is no optional flow, non-zero otherwise
- `sf_init_param_get_flow_name`: the flow name
- `sf_init_param_get_flow_group`: the flow group

Parameters

- `param`: configuration structure
- `kind, number`: defines the flow: input or output, and the flow number

3.2.4 sf_flow_param_init

`sf_flow_param_init` initializes the flow structure:

```
void sf_flow_param_init(sf_flow_param *flow);
```

Returned value

none

Parameters

`flow`: flow structure

3.2.5 sf_flow_param_get_flow

`sf_flow_param_get_flow` gets the flow. It transfers from the application structure to the flow structure: the flow name, group, type, description for a *Data* type flow, picture size for a *Video* flow, and vector size for a *Matrixflow*:

```
void sf_flow_param_init(sf_flow_param *flow, const sf_init_param *param, sf_flow_kind kind, int number);
```

Returned value

none

Parameters

- `flow`: flow structure
- `param`: application structure
- `kind, number`: defines the flow (input/output, the flow number)

3.2.6 sf_flow_param_set_xxx - general

If not defined previously, the general parameters of a flow are set by these three functions:

```
void sf_flow_param_set_name(sf_flow_param *flow, const char *name);
void sf_flow_param_set_group(sf_flow_param *flow, const char *group);
void sf_flow_param_set_type(sf_flow_param *flow, sf_flow_type type);
```

Returned value

none

Parameters

- flow: flow configuration
- name: flow name
- group: flow group (optional)
- type: flow type (SF_TYPE_VIDEO, SF_TYPE_AUDIO, SF_TYPE_DATA, SF_TYPE_VECTOR or SF_TYPE_MATRIX)

3.2.7 sf_flow_param_set_xxx - Video flow

Video flow parameters are set by these two functions:

```
void sf_flow_param_set_subtype(sf_flow_param *flow, int subtype);
void sf_flow_param_set_size(sf_flow_param *flow, int width, int height);
```

Returned value

none

Parameters

- flow: flow configuration
- subtype: required video format (eg. SF_VIDEO_PACKED_RGB)
- width: picture width (optional: default value = 720)
- height: picture height (optional: default value = 480)

3.2.8 sf_flow_param_set_xxx - Audio flow

Audio flow parameters are set by these four functions:

```
void sf_flow_param_set_frequency(sf_flow_param *flow, int frequency);
void sf_flow_param_set_format(sf_flow_param *flow, int format);
void sf_flow_param_set_buffer_size(sf_flow_param *flow, int buffer_size);
void sf_flow_param_set_buffer_offset(sf_flow_param *flow, int buffer_offset);
```

Returned value

none

Parameters

- flow: flow configuration
- frequency: sampling frequency (eg. 44100)
- format: sound format (eg. SF_AUDIO_16BITS_LSB_SIGNED)
- buffer_size: size in sample of each buffer (eg. 1024)
- buffer_offset: offset in sample between two consecutive buffers (eg. 512, ignored for flow creation, using an overlap for windowed data); optional: default value = buffer_size

3.2.9 sf_flow_param_set_xxx - Data flow

If not defined previously, `sf_flow_param_set_description` sets the description of a *Data* flow:

```
void sf_flow_param_set_description(sf_flow_param *flow, DATA_DESC_P);
```

Returned value

none

Parameters

- `flow`: flow configuration
- `DATA_DESC_P`: `DATA_DESC(type)` or `DATA_DESC_WITH_SIZE(type, size)`

3.2.10 sf_flow_param_set_xxx - Vector flow

Vector flow parameters, width and format, are set by the following functions:

```
void sf_flow_param_set_format(sf_flow_param *flow, int format);
void sf_flow_param_set_width(sf_flow_param *flow, int width);
```

Returned value

none

Parameters

- `flow`: flow configuration
- `format`: vector format (eg. `SF_VECTOR_16BITS`)
- `width`: vector width

3.2.11 sf_flow_param_set_xxx - Matrix flow

Matrix flow parameters, size and format, are set by the following functions:

```
void sf_flow_param_set_size(sf_flow_param *flow, int width, int height);
void sf_flow_param_set_format(sf_flow_param *flow, int format);
```

Returned value

none

Parameters

- `flow`: flow configuration
- `width`: matrix width
- `height`: matrix height
- `format`: matrix format (eg. `SF_MATRIX_16BITS`)

3.2.12 sf_init

`sf_init` initializes the system:

```
sf_handle *sf_init(const sf_init_param *param, sf_error *err);
```

Returned value

- `sf_handle` if everything is all right
error code otherwise (eg. `SFE_NO_SERVER` for a server not launched)

Parameters

- `param`: logical name of the application (eg. `Face recognition`)
- `err`: pointer to a `sf_error` structure

3.2.13 sf_flow_subscribe_xxx

- `sf_flow_subscribe_sync` subscribes to a flow with synchronous updating:
`sf_flow_sync *sf_flow_subscribe_sync(sf_handle *handle, const sf_flow_param *flow, int hist_size, sf_error *err);`
- `sf_flow_subscribe_async` subscribes to a flow with asynchronous updating:
`sf_flow_async *sf_flow_subscribe_async(sf_handle *handle, const sf_flow_param *flow, sf_error *err);`
Again, since asynchronous flows are not implemented, this is a request of comment.

Returned value

- flow handle

Parameters

- `handle`: handle returned by `sf_init`
- `flow`: flow characteristics
- `hist_size`: minimum size of the history (number of buffers) kept
- `err`: pointer to a `sf_error` structure

3.2.14 sf_flow_create

- `sf_flow_create` opens a flow in emit mode:
`sf_flow_emit *sf_flow_create(sf_handle *handle, const sf_flow_param *flow, sf_error *err);`

Returned value

- `sf_flow_emit`: flow handle

Parameters

- handle: handle returned by sf_init
- flow: flow characteristics
- err: pointer to a sf_error structure

3.2.15 sf_get_output_bufferxxx

The address of the next local buffer to emit and its number can be retrieved with these two functions:

```
void sf_get_output_buffer(sf_flow_emit *flow, sf_error *err);
void sf_get_output_buffer_number(sf_flow_emit *flow, sf_error *err);
```

Returned value

- sf_get_output_buffer: pointer to the next buffer of data
- sf_get_output_buffer_number: buffer number

Parameters

- flow: flow
- err: pointer to a sf_error structure

3.2.16 sf_send_buffer

sf_send_buffer validates the buffer content and sends it to the flow receptors:

```
void sf_send_buffer(sf_flow_emit *flow, size_t size, sf_error *err);
```

Returned value

none

Parameters

- flow: flow
- size: buffer size if the size is not fixed, 0 otherwise
- err: pointer to a sf_error structure

3.2.17 sf_drop_buffer

sf_drop_buffer drops the next buffer previously requested with sf_get_output_buffer:

```
void sf_drop_buffer(sf_flow_emit *flow, sf_error *err);
```

Returned value

none

Parameters

- flow: flow
- err: pointer to a sf_error structure

3.2.18 sf_get_update_buffer

`sf_get_update_buffer` gets the buffer address for an asynchronous flow

```
const void *sf_get_update_buffer(sf_flow_async *flow, sf_error *err);
```

Returned value

- buffer address

Parameters

- flow: flow
- err: pointer to a sf_error structure

3.2.19 sf_wait_update

`sf_wait_update` waits for the next update to an asynchronous flow:

```
void sf_wait_update(sf_flow_async *flow, sf_error *err);
```

Since asynchronous flows are not implemented, this is a request of comment.

Returned value

none

Parameters

- flow: flow
- err: pointer to a sf_error structure

3.2.20 sf_get_buffer

`sf_get_buffer` gets a local read-only buffer out of the history (blocking function):

```
const void *sf_get_buffer(sf_flow_async *flow, sf_counter *position, sf_error *err);
```

Returned value

- pointer to the buffer
- if a buffer is returned, `position` (see the parameters) is updated with the data packet number when using `sf_hist_xxx`

Parameters

- `flow`: flow to be read
- `position`: pointer to the data packet number or `SF_HIST_XXX` special value
- `err`: pointer to a `sf_error`

`position` can be

- a buffer number:
 - the function gets a reference to this buffer and gives access to it immediately, or blocks to give access later if the buffer belongs to the future, or returns an `sf_error` if the requested buffer has aged out of storage.
- or a special value:
 - `SF_HIST_NEXT`: gives the number of the buffer following the last processed one,
 - `SF_HIST_CURRENT`: gives the number of the buffer that is the latest and not yet processed.

Audio flows use `SF_HIST_NEXT` since no buffer can be skipped with it. For video flows we allow skipped buffers, so `SF_HIST_CURRENT` is sufficient. When it is fast enough, `SF_HIST_CURRENT` is always equivalent to `SF_HIST_NEXT`.

3.2.21 sf_get_buffer_size

`sf_get_buffer_size` gets the real size of a buffer:

```
size_t sf_get_buffer_size(sf_flow_sync *flow, sf_counter position, sf_error *err);
```

Returned value

- real size of the buffer in bytes

Parameters

- `flow`: flow to be read
- `position`: data packet number
- `err`: pointer to a `sf_error`

3.2.22 sf_release_buffer

`sf_release_buffer` releases a local read-only buffer reference:

```
void sf_release_buffer(sf_flow_sync *flow, sf_counter position, sf_error *err);
```

Returned value

none

Parameters

- flow: flow
- position: data packet number
- err: pointer to a sf_error structure

3.2.23 sf_perror

sf_perror prints an error message corresponding to the error object on stderr:

```
void sf_perror(const sf_error *err, const char *comment);
```

Returned value

none

Parameters

- err: error object
- comment: comment to add at the beginning of the message

3.2.24 sf_flow_xxx_metadata

- sf_flow_get_metadata gets metadata on a flow:
`void sf_flow_get_metadata(sf_handle *handle, const char *name, const char *group, const char *metaname, DATA_DESC_P, void *data, sf_error *err);`
- sf_flow_set_metadata sets metadata on a flow:
`void sf_flow_set_metadata(sf_handle *handle, const char *name, const char *group, const char *metaname, DATA_DESC_P, void *data, sf_error *err);`

Returned value

none

Parameters

- handle: handle returned by sf_init
- name: flow name (eg. Video_in)
- group: group name, null for default (eg. Video_1)
- metaname: metadata name
- DATA_DESC_P: DATA_DESC(metadata_type)
- data: pointer to the local value or returned result area
- err: error object

3.2.25 sf_set_blocking_mode

`sf_set_blocking_mode` sets blocking mode on a thread:

```
void sf_set_blocking_mode(sf_handle *handle, pthread_t thread, int blocking,  
sf_error *err);
```

Returned value

none

Parameters

- `handle`: handle returned by `sf_init`
- `thread`: thread handle
- `blocking`: 1 = blocking (default), 0 = non-blocking
- `err`: pointer to an `sf_error` structure

3.2.26 sf_flow_xxx_close & sf_exit

- `sf_flow_sync_close` closes a flow opened in receive mode:

```
void sf_flow_sync_close(sf_flow_sync *flow, sf_error *err);
```

- `sf_flow_emit_close` closes a flow opened in emit mode:

```
void sf_flow_emit_close(sf_flow_emit *flow, sf_error *err);
```

- `sf_exit` quits the system, must be done before process exit, or referenced buffers cannot be freed:

```
void sf_exit(sf_handle handle, sf_error *err);
```

Returned value

none

Parameters

- `flow`: flow
- `handle`: handle returned by `sf_init`
- `err`: pointer to a `sf_error` structure

3.3 Example of use

In order to illustrate the main types and functions of the flow system, this chapter shows the principal sequence of a program using flows. In fact, it gives parts of two programs dealing with the Audio in flow: `sound_play.c` (in receive mode) and `sound_cap.c` (in emit mode).

3.3.1 Type definition

```
sf_handle *me;
sf_error err;
sf_init_param iparam;
sf_flow_param fparam;
static volatile int running;
int fd;
int buffer_size;
int arg;
```

sound_play.c

```
sf_flow_sync *faudio;
```

sound_cap.c

```
sf_flow_emit *faudio;
int resync = 1;
```

3.3.2 Initialize the system parameters

sound_play.c

```
sf_init_param_init(&iparam);
sf_init_param_set_name(&iparam, "Audio Play");
sf_init_param_set_flow_count(&iparam, 0, 1);
sf_init_param_set_flow_name (&iparam, SF_KIND_INPUT, 0, "Audio in");
sf_init_param_set_flow_type (&iparam, SF_KIND_INPUT, 0, SF_TYPE_AUDIO);
sf_init_param_args(&iparam,&argc, argv);
```

sound_cap.c

```
sf_init_param_init(&iparam);
sf_init_param_set_name(&iparam, "Audio Capture");
sf_init_param_set_flow_count(&iparam, 1, 0);
sf_init_param_set_flow_name (&iparam, SF_KIND_OUTPUT, 0, "Audio in");
sf_init_param_set_flow_type (&iparam, SF_KIND_OUTPUT, 0, SF_TYPE_AUDIO);
sf_init_param_args(&iparam,&argc, argv);
```

3.3.3 Initialize the system

```
me = sf_init(&iparam, &err);
if(err.error_code != SFE_OK){
    sf_perror(&err, "Couldn't start system");
    exit(1);
```

```

}
fd = open( "/dev/dsp" , O_RDWR );
if(fd<0){
    perror( "open /dev/dsp" );
    exit(1);
}

```

3.3.4 Initialize the flow parameters

sound_play.c

```

sf_flow_param_init (&fparam);
sf_flow_param_get_flow (&fparam, &iparam, SF_KIND_INPUT, 0);
sf_flow_param_set_frequency (&fparam, 22050);
sf_flow_param_set_format (&fparam, SF_AUDIO_16BITS_LSB_SIGNED);
sf_flow_param_set_buffer_size(&fparam, buffer_size/2);

```

sound_cap.c

```

sf_flow_param_init (&fparam);
sf_flow_param_get_flow (&fparam, &iparam, SF_KIND_OUTPUT, 0);
sf_flow_param_set_frequency (&fparam, 22050);
sf_flow_param_set_format (&fparam, SF_AUDIO_16BITS_LSB_SIGNED);
sf_flow_param_set_buffer_size(&fparam, buffer_size/2);

```

3.3.5 Open a flow in emit mode and emit on it

This is for sound_cap.c.

Open a flow in emit mode

```

faudio = sf_flow_create(me, &fparam, &err);
if(err.error_code!=SFE_OK) {
    sf_perror(&err, "Couldn't create the audio flow");
    exit(1);
}

```

Get the address of the next local buffer to emit

```

running = 1;
while(running) {
    void *buffer = sf_get_output_buffer(faudio, &err);
    if(err.error_code!=SFE_OK) {
        sf_perror(&err, "sf_get_output_buffer");
        exit(1);
    }
}

```

Send the buffer in the locally created flow or drop it

```

read(fd, buffer, buffer_size);
if(running) {
    sf_send_buffer(faudio, 0, &err);
    if(err.error_code!=SFE_OK) {
        sf_perror(&err, "sf_send_buffer");
        exit(1);
    }
}
else {
    sf_drop_buffer(faudio, &err);
    if(err.error_code!=SFE_OK) {
        sf_perror(&err, "sf_send_buffer");
        exit(1);
    }
}
}
}

```

3.3.6 Subscribe to the flow and read its data

This is for sound_play.c.

Subscribe to the flow (with synchronous updating)

```

faudio = sf_flow_subscribe_sync(me, &fparam, 50, &err);
if(err.error_code != SFE_OK) {
    sf_perror(&err, "Couldn't subscribe to the audio flow");
    exit(1);
}

```

Get a read-only local buffer out of the history

```

resync = 1;
running = 1;
while(running) {
    sf_counter pos;
    const void *buffer;
    pos = resync ? SF_HIST_CURRENT : SF_HIST_NEXT;
    buffer = sf_get_buffer(faudio, &pos, &err);
    if(err.error_code != SFE_OK) {
        sf_perror(&err, "sf_get_buffer");
        exit(1);
    }
    resync = 0;
    if(running)
        write(fd, buffer, buffer_size);
}

```

Release this buffer

```
sf_release_buffer(faudio, pos, &err);
```

```
    if(err.error_code != SFE_OK) {
        sf_perror(&err, "sf_release_buffer");
        exit(1);
    }
}
```

3.3.7 Close the flow and quit the system

sound_play.c: close a flow opened in receive mode

```
sf_flow_sync_close(faudio, &err)
```

sound_cap.c: close a flow opened in emit mode

```
sf_flow_emit_close(faudio, &err);
```

Quit the system

```
sf_exit(me, &err);
return 0;
```

3.4 Example

Here is another example, this time for Video in flow. We display the main parts of two C-programs: vcap.c and vdisp.c (some code has been removed for clarity, see actual source code for complete program). The first one illustrates the emit mode since it concerns video capture, and the second one is for the receive mode since it concerns video display.

3.4.1 Video capture: vcap.c

```
#include <sflib.h>

sf_handle *me;
sf_flow_emit *flow;
int skip;
static volatile int running;

int main(int argc, char **argv)
{
    int dfd;
    sf_error err;
    struct sigaction sa;
    sf_init_param iparam;
    sf_flow_param fparam;

    sf_init_param_init      (&iparam);
    sf_init_param_set_name  (&iparam, "LML33 video capture");
    sf_init_param_set_flow_count(&iparam, 0, 1);
    sf_init_param_set_flow_name (&iparam, SF_KIND_OUTPUT, 0,
                                "Video in");
    sf_init_param_set_flow_type (&iparam, SF_KIND_OUTPUT, 0,
                                SF_TYPE_VIDEO);
    sf_init_param_args        (&iparam, &argc, argv);

    me = sf_init(&iparam, &err);
    if(err.error_code != SFE_OK) {
        sf_perror(&err, "Couldn't start system");
        exit(1);
    }

    sf_flow_param_init      (&fparam);
    sf_flow_param_get_flow   (&fparam, &iparam, SF_KIND_OUTPUT, 0);
    sf_flow_param_set_subtype(&fparam, SF_VIDEO_DUAL_JPEG);

    flow = sf_flow_create(me, &fparam, &err);
    if(err.error_code != SFE_OK) {
        sf_perror(&err, "Couldn't create the video in flow");
        exit(1);
    }

    dfd = open("/dev/h33", O_RDWR);
    ...
    if (dfd < 0){
```

```

    perror("Opening of h33 device failed.\n");
    exit(-1);
}

running = 1;

while(running) {
    int i;
    char *buffer = sf_get_output_buffer(flow, &err);
    size_t len = 0;
    if(err.error_code!=SFE_OK) {
        sf_perror(&err, "get_output_buffer");
        exit(1);
    }

    for(i=0;running && (i<skip);i++)
        len = read(fd, buffer, SFI_DJPEG_MAX_SIZE);

    if(running)
        sf_send_buffer(flow, len, &err); /* decrements buffer reference count*/
    else
        sf_drop_buffer(flow, &err);

    if(err.error_code!=SFE_OK) {
        sf_perror(&err, "send_buffer");
        exit(1);
    }
}
sf_flow_emit_close(flow, &err);
sf_exit(me, &err);

return 0;
}

```

3.4.2 Video display: vdisp.c

```

#include <sfplib.h>

/*-----
 struct coords {
 sf_counter frame;
 int x, y, width, height;
 };
-----*/

sf_handle *me;
sf_flow_sync *fvideo, *fcoords;

sf_counter position;

int main(int argc, char **argv)

```

```

{
    sf_init_param iparam;
    sf_flow_param fparam;
    sf_error err;

    gtk_init(&argc, &argv);
    notifier_init();

    sf_init_param_init (&iparam);
    sf_init_param_set_name (&iparam, "Video display");
    sf_init_param_set_flow_count(&iparam, 2, 0);
    sf_init_param_set_flow_name (&iparam, SF_KIND_INPUT, 0,
                                "Video in");
    sf_init_param_set_flow_type (&iparam, SF_KIND_INPUT, 0,
                                SF_TYPE_VIDEO);
    sf_init_param_set_flow_name (&iparam, SF_KIND_INPUT, 1,
                                "coords");
    sf_init_param_set_flow_type (&iparam, SF_KIND_INPUT, 1,
                                SF_TYPE_DATA);
    sf_init_param_set_flow_desc (&iparam, SF_KIND_INPUT, 1,
                                DATA_DESC(struct coords));
    sf_init_param_set_flow_active(&iparam, SF_KIND_INPUT, 1, 0);
    sf_init_param_args (&iparam, &argc, argv);

    me = sf_init(&iparam, &err);
    if(err.error_code != SFE_OK) {
        sf_perror(&err, "Couldn't start system");
        exit(1);
    }

    if(sf_init_param_get_flow_active(&iparam, SF_KIND_INPUT, 1)) {
        sf_flow_param_init (&fparam);
        sf_flow_param_get_flow (&fparam, &iparam, SF_KIND_INPUT, 1);
        fcoords = sf_flow_subscribe_sync(me, &fparam, 1, &err);
        if(err.error_code != SFE_OK) {
            sf_perror(&err, "Couldn't subscribe to the coordinates flow");
            exit(1);
        }
    } else
        fcoords = 0;

    sf_flow_param_init (&fparam);
    sf_flow_param_get_flow (&fparam, &iparam, SF_KIND_INPUT, 0);
    sf_flow_param_set_subtype(&fparam, grayscale ?
                            quarter ? SF_VIDEO_QUARTER_GRAYSCALE
                            : SF_VIDEO_GRAYSCALE :
                            quarter ? SF_VIDEO_QUARTER_PACKED_RGB
                            : SF_VIDEO_PACKED_RGB);

    fvideo = sf_flow_subscribe_sync(me, &fparam, fcoords ? 300 : 1, &err);
    if(err.error_code != SFE_OK) {

```

```
sf_perror(&err, "Couldn't subscribe to the video input flow");
exit(1);
}

VideoOut *vo = VIDEOOUT(video);
for(;;) {
    sf_error err;
    sf_counter pos = SF_HIST_CURRENT;
    const void *buf = sf_get_buffer(fvideo, &pos, &err);
    if(err.error_code == SFE_OK) {
        position = pos;
        videoout_update(vo, buf, 0, 0);
        sf_release_buffer(fvideo, pos, &err);
        if(err.error_code != SFE_OK) {
            sf_perror(&err, "sf_release_buffer");
            exit(1);
        }
    } else if(err.error_code == SFE_TOO_OLD)
        fprintf(stderr, "Dropped frame\n");
    else {
        sf_perror(&err, "sf_get_buffer");
        exit(1);
    }
}
return 0;
}
```


Chapter 4

Conclusion

NIST Smart Flow System architecture has several great advantages. The structure is totally dynamic and the system is network transparent, i.e. independent from the network structure.

Great care has also been taken to reduce memory bandwidth requirements, and make good use of network bandwidth.

Nevertheless, we can't start several communication systems using the same computers at the same time. Actually, the system operates with peer-to-peer communication among the clients implemented with SF_LIB, and does not have closed boundaries for the Smart Flow application.

Since this is an evolutionary architecture, this document will be updated in the future. Of course, any comment about this paper or the communication system is welcome.